基于 B-list 的最大频繁项集挖掘算法

张 昌[1†]，文 凯[1,2]，郑云俊[1]

(1. 重庆邮电大学 通信新技术应用研究中心，重庆 400065; 2. 重庆信科设计有限公司，重庆 401121)

摘 要：针对现有的最大频繁项集挖掘算法挖掘时间过长、内存消耗较大的问题，提出了一种基于构造链表 B-list 的最大频繁项集挖掘算法 BMFI，该算法利用 B-list 数据结构来挖掘频繁项集并采用全序搜索树作为搜索空间，然后采用父等价剪枝技术来缩小搜索空间，最后再结合基于 MFI-tree 的投影策略实现超集检测来提高算法的效率。实验结果表明，BMFI 算法在时间效率与空间效率方面均优于 FPMAX 算法与 MFIN 算法。该算法在稠密数据集与稀疏数据集中进行最大频繁项集挖掘时均有良好的效果。

关键词：最大频繁项集挖掘；深度优先搜索；剪枝技术；超集检测

中图分类号：TP301.6　　doi: 10.3969/j.issn.1001-3695.2017.08.0873

Maximal frequent itemset mining algorithm based on B-list

Zhang Chang[1†], Wen Kai[1,2], Zhengyunjun[1]

(1. *Institute of Applied Communication Technology*, *Chongqing University of Posts & Telecommunications*, *Chongqing* 400065, *China*; 2. *Chongqing Information Technology Designing Co. Ltd*, *Chongqing* 401121, *China*)

Abstract: In order to solve the problems that existing in the maximal frequent itemset mining algorithms, such as the mining time is too long and the memory consumption is too large, this paper presents a maximal frequent itemset mining algorithm BMFI which employs B-list to mining frequent itemsets and employs the whole sequence search tree as the search space, then, the parent equivalence pruning technique is used to reduce the search space. Finally, which combined with the MFI-tree-based projection strategy to achieve superset detection to improve the efficiency of the algorithm. The experimental results show that the performance of BMFI algorithm is superior to FPMAX algorithm and MFIN algorithm in terms of time efficiency and spatial efficiency. The proposed algorithm has good performance when mining the maximal frequent itemset in dense data set and sparse data set.

Key Words: maximal frequent itemsets mining; depth-first search; pruning techniques; superset detection

0 引言

数据挖掘是从大型数据集中挖掘模式的一种计算过程。频繁模式挖掘挖掘经典算法包括 Agrawal 等人[1]在 1994 年提出的 Apriori 算法、Han 等人[2]在 2004 年提出的 FP-growth 算法等。关联规则被广泛应用于“购物篮”分析、电子商务以及医学等领域。在挖掘频繁模式的过程中，也会遇到一些现实的问题，例如在挖掘一个稠密的数据集时，若最小支持度阈值设置较低时，那么产生的频繁模式的数量就会非常庞大。另一方面，当频繁项集较长时，其频繁非空子集的数量也会非常庞大。所以在这种情况之下，将所有的频繁项集都挖掘出来是不可行的。

最大频繁项集是频繁项集的一种紧凑表示，具备更高的挖掘效率[3]。若一个频繁项集 S 满足其真超集都是不频繁的，那么称 S 是最大频繁项集。近年来有很多的专家学者都对最大频繁项集挖掘算法进行了深入研究，Bayardo 等人[4]提出了 MaxMiner 算法，该算法采用了广度优先搜索策略和基于动态排序的超集检测方法。Agrawal 等[5]提出了 DepthProject 算法，该算法不仅采用了深度优先搜索策略和基于动态排序的超集检测方法之外，还采用了基于桶计算的的项集计算方法。Burdick 等人[6]提出了 MAFIA 算法，该算法采用了 DFS 策略，并且提出了结合 DFS 的剪枝策略 PEP、FHUT、MFIHUT 和 Dynamic Reordering，同时还采用了一种高效的超集检测策略 LMFI。Grahne 等人[7]提出了 FPMAX 算法，该算法先根据事务数据库建立 FP-tree，再采用基于 MFI-tree 的投影策略实现超集检测，从而提升了算法的性能。沈戈晖等[8]在深度优先搜索的基础之上，引入了基于 N-list 最大频繁项集挖掘算法，该算法利用了 N-list 的高效压缩率与高效求交集的特点，并采用了搜索空间的剪枝策略与超集检测方法提升了挖掘效率，但该算法在数据

作者简介：张昌（1993-），男，湖北孝感人，硕士研究生，主要研究方向为数据挖掘、大数据、云计算（420333249@qq.com）；文凯（1972-），男，正高级工程师，博士，主要研究方向为通信新技术应用研究、大数据；郑云俊（1992-），男，硕士研究生，主要研究方向为大数据、推荐系统.

集稀疏的情况下，算法性能有待提高。林晨等[9]提出了基于 Nodeset 的最大频繁项集挖掘算法—MFIN 算法，该算法采用了 POC-tree 结构对

节点编码，用全序搜索作为搜索空间，然后结合了父等价剪枝等技术对数据进行剪枝处理，提高了算法的效率。当数据量较大时该算法也会存在运行时间过长和内存消耗大的问题，挖掘效率有待提高。在 2015 年 Deng 等[11]提出了基于 N-list 的 PrePost+算法，该算法运用的是子孙-祖父等价的剪枝技术，使用这种剪枝技术在挖掘频繁项集方面比 FIN 算法与 PrePost 算法性能更好。

针对现有的最大频繁项集挖掘算法挖掘时间过长、内存消耗较大的问题，本文提出了一种基于构造链表 B-list[12]的最大频繁项集挖掘算法 BMFI。该算法在深度优先算法的基础上利用 B-list 高压缩率和高效求交集并能快速计算项集支持度的特点，然后采用父等价剪枝技术来缩小搜索空间。在多个数据集上运行的实验结果表明 BMFI 比 FPMAX 算法和 BMFI 算法在时间效率与空间消耗方面都有明显的提升，在稠密数据集下时优化效果更为明显。

1　相关知识

1.1　基本概念

设 $I=\{i_1,i_2,\cdots,i_m\}$ 是 n 个不同项组成的集合，包含 m（0≤m≤n）个项的集合称为项集。

定义 1　若项集 X 的支持度大于或等于用户规定的最小支持度阈值(minSup)，则项集 X 是频繁项集。

定义 2　若频繁项集 X 的任何超集 Y 都不是频繁项集，则 X 是最大频繁项集。

1.2　TB-tree

针对现有的频繁项集挖掘算法中 PPC-tree[13]，POC-tree[10]等建树复杂、效率较低等问题。本文引入了一种 TB-tree[12]的结构，TB-tree 是由一个 root 根节点和其项目前缀子树构成，项目前缀子树中的每个节点由五个部分组成，分别是 item-name, count, parent-pointer, start-build 和 finish-build。TB-tree 的结构与 PPC-tree 相似，但 PPC-tree 需要前序和后序两次遍历得到每个节点的 pre-order 和 post-order 来获取各节点信息。POC-tree 虽然只需要前序或后序一种编码，结构相对 PPC-tree 更加的简洁，但仍需要一次遍历才能获得各节点信息。TB-tree 各节点的 start-build 与 finish-build 依据各节点在 TB-tree 的构建顺序中得到，即每个节点的信息在树构建完成时就已获得，所以比 PPC-tree 与 POC-tree 的建立更具有时间效率。

如表 1 所示是事务数据库 DB，BMFI 算法首先扫描事务数据集，并将事务集中的事务按支持度降序排列，删除掉支持度低于 minSup 的事务。然后根据排序后的事物集，构造 TB-tree。构造树的优势在于各节点信息在构建树的过程中就已获得，不需再对树进行遍历，详细的构造过程参考文献[12]。TB-tree 的构建过程见算法 1，图 1 是对应数据库 DB 构造完成的 TB-tree。

表 1　事务数据集 DB

ID	Items	Order frequent items
1	e,a,c	c,e,a
2	a,b,c,g,f	b,c,f,a
3	e,c,b,f,i	b,c,e,f
4	b,h,c	b,c
5	b,f,e,d	b,e,f

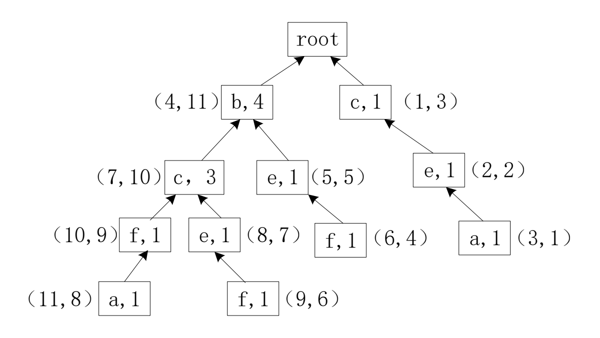


图 1　DB 对应的 TB-tree（minSup=0.4）

算法 1　构建 TB-tree

输入：事务数据库 DB；

输出：TB-tree，L_1。

扫描事务数据库 *DB* 得到频繁 1-项集 L_1，并按项目支持度降序排列,将 *DB* 中的事务的项按 L_1 的顺序排列,创建根节点 root，初始化全局变量 *star t*= 0，*finish* = 0;

for each different first item *p* in *DB* do

Call BuildTree(*p*,*Node*)

end for

end

function buildTree(*p*,*parent*)

Let T_P be a list of transactions in *DB* which contain prefix *p*

Creat node *N* :

N.item-name = *item-name* of the last item in *p*

N.count = count of transactions in T_P,

N.parent = parent

N.start-build = ++start

for each different first item *q* in T_P do

Call buildTree($p \cup q$, *N*)

end for

N.finish-build = ++finish

end function

1.3　B-list 结构

定义 3 B-info-code。在 TB-tree 中的每个节点 N[(*N.start-build*，*N.finish-build*)，sup(N)]即节点 N 的 B-info-code。其中 *start-build* 和 *finish-build* 分别是改点的开始构造与结束构造的顺序。

性质 1　对于 N_1 和 N_2 两个节点而言，当且仅当 N_1.

start-build< N_2.start-build 与 N_1.finish-build> N_2.finish-build 同时成立时 N_1 才是 N_2 的祖先节点。

例 1 如图 1 中的 TB-tree 所示，节点 B 与 A 的 B-info-codes 分别为 B_b={[(4,11),4]}和 B_c={[(7,10),3]}，B_b.*start-build* <B_c.start -build(4<7)，B_b. finish-build>B_c. finish-build(11>10)。所以 b 是 a 的祖先节点。任意两节点都能根据该性质通过 start-build 与 finish-build 信息来判定是否存在祖先-子孙关系。

定义 4 1-项集的 B-list。对于已构建完成的 TB-tree 与节点 N，其中名为 N 的节点的所有 B-info-code 的集合成为节点 N 的 B-list，记为 BL_N。

例 2 如图 1 所示的 TB-tree 中，节点 B 的 B-list 是 BL_B={[(2,1),1],[(7,4),2]}。

定义 5 2-项集的 B-list。假设一个 2-项集{i_1,i_2}，且 i_1 在 i_2 之前。i_1 的 B-list 为{[(s_{11}，f_{11})，c_{11}]，[(s_{12}，f_{12})，c_{12}]，…[(s_{1n}，f_{1n})，c_{1n}]}，i_2 对应的 B-list 为{[(s_{21}，f_{21})，c_{21}]，[(s_{22}，f_{22})，c_{22}]，…[(s_{2m}，f_{2m})，c_{2m}]}，则生成 2-项集{i_1,i_2}的 B-list 要遵循以下规则。

a)存在[(s_{1i}，f_{1i})，c_{1i}]（$1\leqslant i\leqslant n$）是[(s_{2j}，f_{2j})，c_{2j}]（$1\leqslant j\leqslant m$）的祖先，就将[(s_{1i}，f_{1i})，c_{2j}]添加到{i_1,i_2}的 B-list 中。

b)对于{i_1,i_2}的 B-list 中(s，f)相同的节点信息，可以将支持度直接相加，合并为[(s，f)，c_1+c_2+…+c_n]。

定义 6 k-项集的 B-list。(k≥3）假设两个 k-1 项集分别为 N=$i_1i_2\cdots i_{k-1}$，M= $i_1i_2\cdots i_{k-1}$，N 的 B-list 记做{[(s_{11}，f_{11})，c_{11}]，[(s_{12}，f_{12})，c_{12}]，…[(s_{1n}，f_{1n})，c_{1n}]},M{[(s_{21}，f_{21})，c_{21}]，[(s_{22}，f_{22})，c_{22}]，…[(s_{2m}，f_{2m})，c_{2m}]}，其相互连接与合并生成 k-项集的规则与定义 4 中的 2-项集生成过程类似（参考文献[12]有详细过程），求交集的算法如算法 2 所示。

算法 2 Intersection

Intersection(BL_cur，BL_HUT，threshold)

输入：项集 X 与 Y 的 B-list，最小支持度阈值 threshold。

输出：两个 B-list 的交集以及支持度计数。

```
R←Ø
Let i=0,j=0,Rsupport=0
C1 是 BL_cur 的支持度计数，C2 是 BL_HUT 的支持度计数
Let s=1;
While(i< BL_cur.size and j< BL_HUT.size) do
if(BL_cur[i].start-build > BL_HUT[j].start-build) then
if(BL_cur[i].finish-build< BL_HUT[j].finish-build)
then
s=0
if(R.size>0 and R[R.size-1].start-build= BL_HUT[j].start-build) then
R[R.size-1].support+= BL_cur[j].support
else
R←B-info-code{BL_HUT[j].start-build,BL_HUT[j].finish-build,
BL_cur[i].support}
end if
Rsupport += BL_cur[i++].support
else
C2= C2-s* BL_HUT[j++].support
s=1
end if
else
C1= C1-BL_cur[i++].support
s=1
end if
if(C1 <threshold or C2 <threshold ) then
return NULL
end if
end while
return R and RSupport
```

1.4 BMFI 算法

本节提出了一种基于 B-list 的最大频繁项集挖掘算法—BMFI 算法，BMFI 算法首先根据事务数据及构建 TB-tree 同时生成 B-lis 然后根据 B-list 的连接原则将两个频繁 k-1 项集连接成频繁 k 项集同时能得到其支持度。本算法采用了全序搜索树[6]为搜索空间，然后采用深度优先搜索（DFS）策略来遍历整个全序搜索树。再采用父等价剪枝策略来缩小搜索空间，最后采用基于 MFI-tree 的投影策略实现超集检测来提高算法的效率。

1.4.1 基于 B-ist 的深度优先搜索策略

假设一个完整的事务数据库，其中包含的数据集按照字典序进行排列，即若项目 i 出现在项目 j 之前，可以用 $i\leqslant_L j$ 表示，如图 2 所示是由 4 个项目（$a\leqslant_L b\leqslant_L c\leqslant_L d$）构成的字典搜索树结构。该结构对于每个节点而言需要与右侧的每个节点分别做并集，从而生成该节点的子集。BMFI 算法采用将 2 个（k-1）项集的 B-list 求交集生成 k-项集的 B-list。

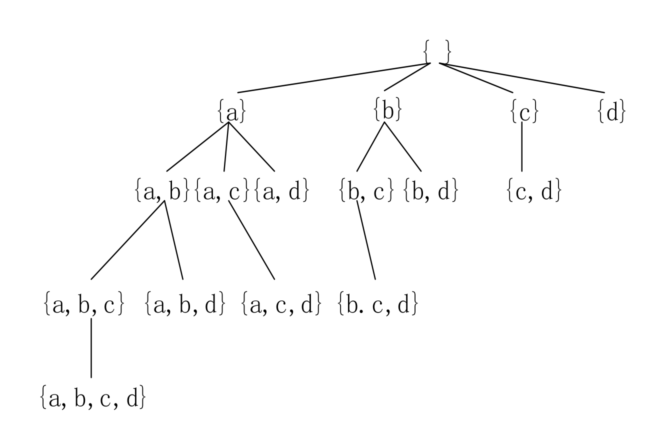


图 2 字典搜索树

定义 7[6] 在全序搜索树中的节点 C，该节点的项集记为 C.head，从该节点可扩展的项的集合记为 C.tail，C.tail 中会包含多个扩展项，每个扩展项集都是 C 节点的扩展 1-项集。C 节点右侧同父节点的集合记 C.sibling+。

例 3 如图 2 所示，假设{b，c}表示节点 C，则根据定义

6，C.head={b，c}，C.tail={d}，则 C.HUT={b，c，d}，C.sibling+={{b，d}}。d 称为节点 C 的 1-扩展信息。

深度优先搜索策略的核心思想是检测每个节点的 C.tail 中的每个 1-扩展项 i。计算 C.head∪i 的支持度，若支持度均小于 minSup 时，此时在频繁模式树中 C 就是一个叶节点。此时检测 C.head 是否是最大频繁项集 MFI 中任意集合的子集。若不是，就将 C 加入到 MFI。若 C.head∪i 支持度均大于 minSup，则继续向下递归。这里参考了文献[6]中将 DFS 策略与其他剪枝策略相结合的思想和文献[8]中将 N-list 与 DFS 相结合的思想，从而引入了一种基于 B-list 的 DFS 算法，由于 B-List 的高压缩率以及 TB-tree 的高效建树的特点，所以在挖掘最大频繁项集时效率会更高。算法如下所示：

算法 3 基于 B-list 的 DFS 算法

输入： 事务数据库 DB 和 minSup；

输出： DB 的所有的 MFI。

```
MFI←Ø
root.head←Ø
root.tail←DB 频繁项的集合(按照项的支持度升序排列)
在构建 TB-tree 树完成之后，得到了所有的频繁 1 项集的 B-list,记为 BL
Call DFS_based_BList(root,NULL,NULL)
Function DFS_based_BList(Current node C,C's BList BL_cur,C.sibling+s'BList[] BLS)
k←C.head 中支持度最大的项
for each item i in C.tail
if(BL_cur==NULL)
BL_child[i]=BL1[i]
else
C_HUT←C.head∪{i}-{k}
BL_HUT←BLS 中记录 C_HUT 的元素
BL_child[i]=BL_intersection(BL_cur,BL_HUT)
endif
endfor
for each item i in C.tail
Cn..head←C.head∪{i}
Cn..tail←{j⊆C.tail|i≤Lj}
C_ HUT ←C.head∪{i}-{k}
BLS_child←BL_child
if(BL_child[i].support≥minSup)
DFS_based_BList(Cn..head,BL_child[i],BLS_child)
endif
endfor
if(C is a leaf and C.head is not in MFI )do
MFI←MFI∪C.head
endif
```

1.4.2 优化策略

性质 2 设 X∈C.head，y∈C.tail，若存在 X 的事务集与 X∪{y}完全相同，即 t(X)=t(X∪{y})，那么∀S∈C.tail，都有 X∪S 与 X∪{y}∪S 的支持度相同。

证明 由于 t(X)=t(X∪{y}),则 Sup(X)=Sup(X∪{y})，所以任意包含 X 的事务必然包含 y。所以，包含 S 的任意项集也必然包含 y，所以 Sup(X∪S)=Sup(X∪{y}∪S)。

BMFI 算法根据性质 2 采用了父等价剪枝技术来缩小搜索空间。当搜索 C 节点时，若遇到性质 2 中的情况，就可以将 y 从 C.tail 中删除，并放入 C.head，这样会提高挖掘的效率并不会影响 MFI 挖掘的准确性。

性质 3 若字典搜索树的节点 C 没有右侧的邻居节点，并且节点 C 不是已经挖掘的最大频繁项集的子集，则该项集是最大频繁项集。

证明 若 C 节点存在右侧的兄弟节点，那么节点 C 所表示的项集就有可能是该右侧兄弟节点所表示项集的子集，所以 C 节点所表示的项集不一定是已表示最大频繁项集的子集。同理，若某节点表示的项集是已挖掘到的最大频繁项集的子集，那么该节点表示的项集不是最大频繁项集。

基于性质 3，本文引入了 Grahne G 等[7]提出的基于 MFI-tree 投影策略来实现超集检测。对于自底向上深度优先遍历 FP-tree 树空间过程中发现的单一路径 FP-tree 头表项集与头项集中的项目构成的项目集，当头表不包含当前项，那么必定不会不存在超集，只有在当前节点路径上存在于与项集重合的部分，那么 MFI-tree 就存在该项集的超集，即实现超集检测。若该项集在 MFI 不存在超集，该项集是最大频繁项集并加入到 MFI。算法 4 中有比较详细的过程。

算法 4 基于 MFI-tree 投影策略的超集检测

输入：进行剪枝后按支持度递减排序后的项集 M

输出：最大频繁模式树

```
int len=M.length;
if(!headerTable.contains(M[len-1]))
return false;
else
BMFINode
node=headerTable.get(M[M.len-1])
While(node!=null）
BMFINode high=node;
While(high!=null)
if(high.label==M[len-1])
len--;
high=high.parent;
if(len==0) //此时就存在超集
return true;
node=node.sibling;
len=M.length;
```

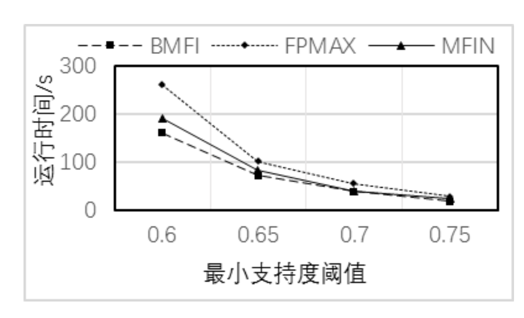
return false;

2　实验结果及分析

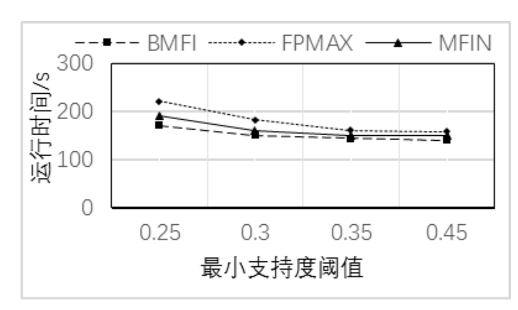
由于 FPMAX 算法在不同的数据集中都保持着良好的性能。所以通过比较 BMFI 与 FPMAX 算法和 MFIN 算法在不同数据集上挖掘最长频繁项集所用的时间来验证 BMFI 算法的有效性。实验所有的程序均用 C/C++编写，实验环境为 Inter(R) Core(TM) i5 CPU M330 @ 3.1GHz CPU，4 GB 内存，64 位 Windows 10 操作系统。在该环境下实现了 FPMAX 算法与 BMFI 算法和 MFIN 算法。实验所使用的是 Pumsb、Retail 数据集以及由 IBM 数据生成器生成的人工数据集 T10I4D100K（各数据集参数如表 2 所示）。通过改变最小支持度来进行频繁模式的挖掘，然后对比分析了算法的运行时间以及内存消耗情况。运行时间对比如图 3 所示，内存消耗情况对比如图 4 所示。

表 2　数据集参数

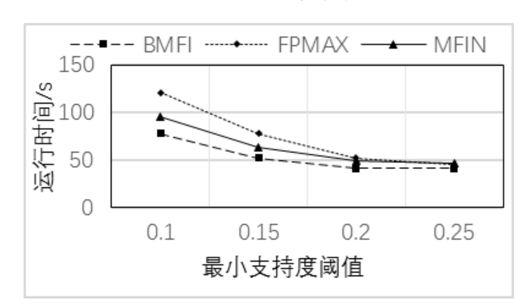
数据集	项数	平均长度	事务数目
Pumsb	7117	74	49046
Retail	16470	10.3	88162
T10I4D100K	870	10.1	100000



（a）Pumsb 数据集

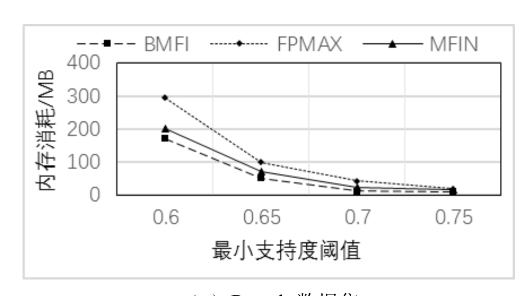


（b）Retail 数据集

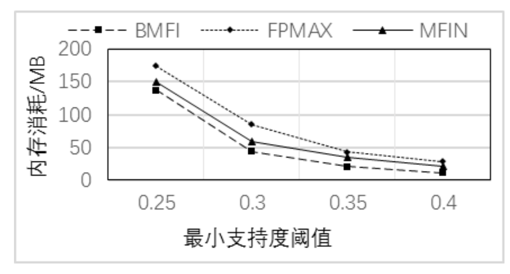


（c）T10I4D100K 数据集

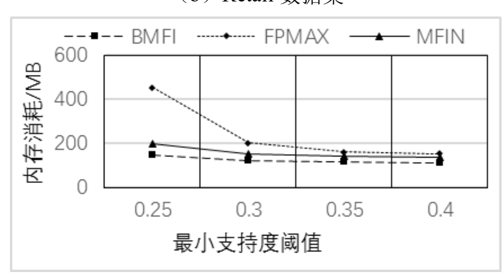
图 3　运行时间对比



（a）Pumsb 数据集



（b）Retail 数据集



（c）T10I4D100K 数据集

图 4　内存消耗对比

本实验将 BMFI 算法与 FPMAX 算法和 MFIN 算法在不同数据集下的挖掘结果进行了分析，首先三种算法在同一数据集同一支持度下挖掘出了最大频繁项集的内容与数量是相同的，表明本算法是正确的。Pumsb、Retail、T10I4D100K 数据集分别代表着稠密型、稀疏型的数据集和 IBM 数据生成器生成的人工数据集，如图 3 所示，三种算法的运行时间都随支持度增加而减少，在 Retail 数据集中上运行时，虽然三种算法在各最小支持度阈值情况下的运行时间相差不大，但在支持度阈值较小时，BMFI 算法的运行时间上有较显著的优势。在 T10I4D100K 数据集上运行时，当最小支持度阈值为 0.1 时,其运行时间相对于 FP-max 算法提升了近 1 倍,随着最小支持度阈值的升高, BMFI 算法运行时间的优势逐渐减小。如图 4 所示,在内存的消耗方面，在不同的数据集中，三种算法的内存消耗也是随着最小支持度阈值的升高而减小，在支持度阈值较小时，BMFI 算法的内存消耗都明显少于 FP-MAX 算法与 MFIN 算法。结果表明 BMFI 算法在不同类型数据集中都有着较高的时间效率与空间效率。

3　结束语

本文提出了一种新的最大频繁模式挖掘算法—BMFI，该算法采用了基于 B-list 的高压缩率以及高效的求交集的方法实现支持度的快速的计算，同时采用父等价剪枝技术来缩小搜索空间，最后再结合基于 MFI-tree 的投影策略实现超集检测，从而来提高算法的效率。结果表明,与 FPMAX 算法和 MFIN 算法相

比，BMFI 算法具有较显著的性能优势。但随着如今互联网大数据时代的到来，BMFI 算法还需要再进行优化，例如在 BMFI 的剪枝优化策略方面还有优化空间，这将是接下来需要思考的问题。

参考文献：

[1] Agrawal R, Srikant R. Fast algorithms for mining association rules [C]// Proc of the 20th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers, 1994: 487-499

[2] Han J, Pei J, Yin Y, et al. Mining frequent patterns without candidate generation: a frequent-pattern tree approach [J]. Data Mining & Knowledge Discovery, 2004, 8 (1): 53-87.

[3] Aggarwal C C, Han J. Frequent pattern mining [M]. [S. l] : Springer International Publishing, 2014

[4] Jr Bayardo R J. Efficiently mining long patterns from databases [J]. ACM Sigmod Record, 1998, 27 (2): 85-93.

[5] Agarwal R C, Aggarwal C C, Prasad V V V. Depth first generation of long patterns [C]// Proc of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. NewYork: ACM Press, 2000: 108-118.

[6] Burdick D, Calimlim M, Flannick J, et al. MAFIA: a maximal frequent itemset algorithm [J]. IEEE Trans on Knowledge and Data Engineering, 2005, 17 (11): 1490-1504.

[7] Grahne G, Zhu J. High performance mining of maximal frequent itemsets [C]// Proc of the 6th International Workshop on High Performance Data Mining. 2003.

[8] 沈戈晖，刘沛东，邓志鸿. NB-MAFIA：基于 N-List 的最长频繁项集挖掘算法 [J]. 北京大学学报：自然科学版, 2016, 52 (2): 199-209.

[9] 林晨，顾君忠. 基于 Nodeset 的最大频繁项集挖掘算法 [J]. 计算机工程, 2016, 42 (12): 204-207, 216.

[10] Deng Z H, Lv S L. Fast mining frequent itemsets using Nodesets [J]. Expert Systems with Applications, 2014, 41 (10): 4505-4512.

[11] Deng Z H, Lv S L. PrePost+: an efficient N-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning [J]. Expert Systems with Applications, 2015, 42 (13): 5424-5432.

[12] Dam T L, Li K, Fournier-Viger P, et al. An efficient algorithm for mining top-rank-k frequent patterns [J]. Applied Intelligence, 2016, 45 (1): 9

[13] Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists [J]. Science China Information Sciences, 2012, 55 (9): 2008-2030.